

Restructuring Compressed Texts without Explicit Decompression

Keisuke Goto¹ Shirou Maruyama¹ Shunsuke Inenaga¹ Hideo Bannai¹
Hiroshi Sakamoto² Masayuki Takeda¹

¹ Kyushu University, Japan

² Kyushu Institute of Technology, Japan

shiro.maruyama@i.kyushu-u.ac.jp

{keisuke.gotou,bannai,inenaga,takeda}@inf.kyushu-u.ac.jp

hiroshi@ai.kyutech.ac.jp

Abstract

We consider the problem of *restructuring* compressed texts without explicit decompression. We present algorithms which allow conversions from compressed representations of a string T produced by any grammar-based compression algorithm, to representations produced by several specific compression algorithms including LZ77, LZ78, run length encoding, and some grammar based compression algorithms. These are the first algorithms that achieve running times polynomial in the size of the compressed input and output representations of T . Since most of the representations we consider can achieve exponential compression, our algorithms are theoretically faster in the worst case, than any algorithm which first decompresses the string for the conversion.

1 Introduction

Data compression is an indispensable technology for the handling of large scale data available today. The traditional objective of compression has been to save storage and communication costs, whereas actually using the data normally requires a decompression step which can require enormous computational resources. However, recent advances in *compressed string processing* algorithms give us an intriguing new perspective in which compression can be regarded as a form of pre-processing which not only reduces space requirements for storage, but allows efficient processing of the strings, including compressed pattern matching [25, 40, 10, 11], string indices [33, 7, 22], edit distance and its variants [9, 15, 38], and various other applications [12, 16, 14, 30, 2]. These methods assume a compressed representation of the text as input, and process them without explicit decompression. An interesting property of these methods is that they can be theoretically – and sometimes even practically – faster than algorithms which work on an uncompressed representation of the same data.

The main focus of this paper is to develop a framework in which various processing on strings can be conducted entirely in the world of compressed representations. A primary tool for this objective is *restructuring*, or conversion, of the compressed representation. Key results for this problem were obtained independently by Rytter [36] and Charikar *et al.* [5]: given a non-self referential LZ77-encoding of size n that represents a string of length N , they gave algorithms for constructing a balanced grammar of size at most $O(n \log(N/n))$ in output linear time. The size of the resulting grammar is an $O(\log(N/g))$ approximation of the smallest grammar whose size is g . Grammars are generally easier to handle than the LZ-encodings, for example, in compressed pattern matching [11], and this result is the motivational backbone of many efficient algorithms on grammar compressed strings.

Our Results: In this paper, we present a comprehensive collection of new algorithms for restructuring to and from compressed texts represented in terms of run length encoding (RLE), LZ77 and LZ78 encodings, grammar based compressor RE-PAIR and BISECTION, edit sensitive parsing (ESP), straight line programs (SLPs), and admissible grammars. All algorithms achieve running times polynomial in the size of the compressed input and output representations of the string. Since (most of) the representations we consider can achieve exponential compression, our algorithms are theoretically faster in the worst case, than any algorithm which first decompresses the string for the conversion. Figure 1 summarizes our results. Our algorithms immediately allow the following applications to be solvable in polynomial time in the compressed world:

Dynamic compressed texts: Although data structures for *dynamic* compressed texts have been studied somewhat in the literature [3, 13, 4, 27, 35, 23], grammar based or LZ77 compression have not been considered in this perspective. It has recently been argued that for *highly repetitive* strings, grammar based compression and LZ77 compression algorithms are better suited and achieve better compression [7, 22].

Modification of the grammar corresponding to edit operations on the string can be conducted in $O(h)$ time, where h is the height of the grammar. (Note that when the grammar is balanced, $h = O(\log N)$ even in the worst case.) However, these modifications are *ad-hoc*, and do not assure that the resulting grammar is a *good* compressed representation of the string, and repeated edit operations will inevitably cause degradation on the compression ratio. Note that previous work of Rytter and Charikar *et al.* are not sufficient in this respect: their algorithms can balance an arbitrary grammar, but they must be given an LZ-encoding of the modified string in order for the grammar to be small.

Post-selection of compression format: Some methods in the field of data mining and machine learning utilize compression as a means of detecting and extracting meaningful information from

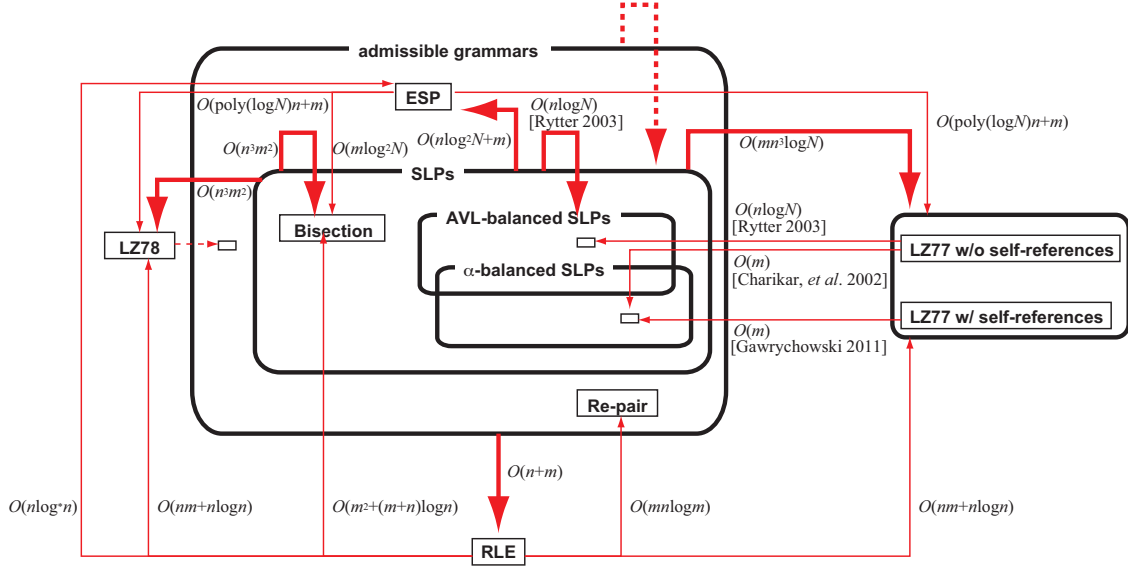


Figure 1: Summary of transformations between compressed representations. The label of each arc shows the time complexity of each transformation, where n and m are respectively the input and output sizes of each transformation, and N is the length of the uncompressed string. The broken arcs mean naive $O(n)$ -time transformations. Complexities without references are results shown in this paper.

string data [8, 6]. Compression of a given string is achieved by exploiting various regularities contained in the string, and since different compression algorithms capture different regularities, the usefulness of a specific representation will vary depending on the application. As it is impossible to predetermine the *best* compression algorithm for all future applications, conversion of the representation is an essential task.

For example, the normalized compression distance (NCD) [6] between two strings X and Y with respect to compression algorithm A is defined by the values $C_A(XY)$, $C_A(X)$, and $C_A(Y)$ which respectively denote the sizes of the compressed representation of strings XY , X , and Y when compressed by algorithm A . Restructuring enables us to solve, in the compressed world, the problem of calculating the NCD with respect to some compression algorithm, given strings which were compressed previously by a (possibly) different compression algorithm.

2 Preliminaries

2.1 Notations

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The length of a string S is denoted by $|S|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $S = XYZ$, X , Y and Z are called a *prefix*, *substring*, and *suffix* of S , respectively. The set of all substrings of a string S is denoted by $Substr(S)$. The i -th character of a string S is denoted by $S[i]$ for $1 \leq i \leq |S|$, and the substring of a string S that begins at position i and ends at position j is denoted by $S[i : j]$ for $1 \leq i \leq j \leq |S|$. For convenience, let $S[i : j] = \varepsilon$ if $j < i$. For any strings S and P , let $Occ(S, P)$ be the set of occurrences of P in S , i.e., $Occ(S, P) = \{k > 0 \mid S[k : k + |P| - 1] = P\}$.

We shall assume that the computer word size is at least $\log |S|$, and hence, values representing lengths and positions of S in our algorithms can be manipulated in constant time.

2.2 Suffix Arrays and LCP Arrays

The suffix array SA [28] of any string S is an array of length $|S|$ such that $SA[i] = j$, where $S[j : |S|]$ is the i -th lexicographically smallest suffix of S . Let $lcp(S_1, S_2)$ is the length of the longest common prefix of S_1 and S_2 . The lcp array of any string S is an array of length $|S|$ such that $LCP[i]$ is $lcp(S[SA[i-1] : |S|], S[SA[i] : |S|])$ for $2 \leq i \leq |S|$, and $LCP[1] = 0$. The suffix array for any string S can be constructed in $O(|S|)$ time (e.g. [17]) assuming an integer alphabet. Given the string and suffix array, the lcp array can also be calculated in $O(|S|)$ time [19].

2.3 Run Length Encoding

Definition 1 *The Run-Length (RL) factorization of a string S is the factorization f_1, \dots, f_n of S such that for every $i = 1, \dots, n$, factor f_i is the longest prefix of $f_i \cdots f_n$ with $f_i \in F$, where $F = \bigcup_{a \in \Sigma} \{a^p \mid p > 0\}$.*

We note that each factor f_i can be written as $f_i = a_i^{p_i}$ for some symbol $a_i \in \Sigma$ and some integer $p_i > 0$ and the repeating symbols a_i and a_{i+1} of consecutive factors f_i and f_{i+1} are different. The output of RLE is a sequence of pairs of symbol a_i and integer p_i . The number of distinct bigrams occurring in S is at most $2n - 1$, since these are $\{a_i a_i \mid 1 \leq i \leq n\} \cup \{a_i a_{i+1} \mid 1 \leq i < n\}$.

2.4 LZ Encodings

LZ encodings are dynamic dictionary based encodings. There are two main variants for LZ encodings, LZ78 and LZ77.

The LZ78 encoding [42] has several variants. One most popular variant would be the LZW encoding [39], which is based on the LZ78 factorization defined below.

Definition 2 (LZ78 factorization) *The LZ78-factorization of a string S is the factorization f_1, \dots, f_n of S where for every $i = 1, \dots, n$, factor f_i is the longest prefix of $f_i \cdots f_n$ with $f_i \in F_i$, where F_i is defined by $F_1 = \Sigma$ and $F_{i+1} = F_i \cup \{f_i f_{i+1}[1]\}$.*

The output is the sequence of IDs of factors f_i in F_i . We note that F_i can be recovered from this sequence and thus is not included in the output.

The LZ77 encoding [41] also has many variants. The LZSS encoding [37] is based on the LZ77 factorization below. The LZ77 factorization has two variations depending upon whether self-references are allowed.

Definition 3 (LZ77 factorization w/o self-references) *The LZ77-factorization without self-references of a string S is the factorization f_1, \dots, f_n of S such that for every $i = 1, \dots, n$, factor f_i is the longest prefix of $f_i \cdots f_n$ with $f_i \in F_i$, where $F_i = \text{Substr}(f_1 \cdots f_{i-1}) \cup \Sigma$.*

Definition 4 (LZ77 factorization w/ self-references) *The LZ77-factorization with self-references of a string S is the factorization f_1, \dots, f_n of S such that for every $i = 1, \dots, n$, factor f_i is the longest prefix of $f_i \cdots f_n$ with $f_i \in F_i$, where $F_i = \text{Substr}(f_1 \cdots f_{i-1} f'_i) \cup \Sigma$, where f'_i is the prefix of f_i obtained by removing the last symbol.*

The LZSS is based on the LZ77 with self-references and its output is a sequence of pointers to factors f_i .

2.5 Grammar-based compression methods

An *admissible grammar* [20] is a context-free grammar that generates a single string.

2.5.1 Re-pair

Starting with $w_1 = S$, we repeat the following until no bigrams occur more than once in w_i : we find a most frequent bigram γ_i in the string w_i , and then replace every non-overlapping occurrence of γ_i in w_i with a new variable X_i to obtain string w_{i+1} . Let r be the number of iterations. The resulting grammar has the production rules of $\{X_i \rightarrow \gamma_i\}_{i=1}^r \cup \{X_{r+1} \rightarrow w_{r+1}\}$.

Theorem 5 ([5]) *For any string S of length N , Re-pair constructs in $O(N)$ time an admissible grammar of size $O(g(N/\log N)^{2/3})$, where g is the size of the smallest grammar that derives S .*

2.5.2 Bisection

The Bisection algorithm [20, 21] constructs a grammar that can be described recursively as follows: the variable representing string S ($|S| \geq 2$) is derived by the rule $X \rightarrow YZ$, with $|Y| = 2^k$ and $|Z| = |X| - 2^k$, where k is the largest integer s.t. $2^k < |X|$. The production rules for $S[1 : 2^k]$ and $S[2^k + 1 : |S|]$ are defined recursively. Whenever $S[i : i + q - 1] = S[j : j + q - 1]$ for some $i, j, q \geq 1$ which appear in the above construction, the same variable is to be used for deriving these substrings.

Theorem 6 ([5]) *For any string S of length N , Bisection constructs an admissible grammar of size $O(g(N/\log N)^{1/2})$, where g is the size of the smallest grammar that derives S .*

2.6 Edit-sensitive parsing (ESP)

A string a^k ($k \geq 2$) is called a repetition of symbol a , and a^+ is its abbreviation. We let $\log^{(1)} n = \log n$, $\log^{(i+1)} n = \log \log^{(i)} n$, and $\log^* n = \min\{i \mid \log^{(i)} n \leq 1\}$. For example, $\log^* n \leq 5$ for any $n \leq 2^{65536}$. We thus treat $\log^* n$ as a constant for sufficiently large n .

We assume that any context-free grammar G is *admissible*, i.e., G derives just one string and for each variable X , exactly one production rule $X \rightarrow \alpha$ exists. The set of variables is denoted by $V(G)$, and the set of production rules, called dictionary, is denoted by $D(G)$. We also assume that $X \rightarrow \alpha \in D(G)$ and $Y \rightarrow \alpha \in D(G)$ implies $X = Y$ because one of them is unnecessary. We use V and D instead of $V(G)$ and $D(G)$ when G is omissible. The string derived by D from a string $S \in (\Sigma \cup V)^*$ is denoted by $S(D)$. For example, when $S = aYY$ and $D = \{X \rightarrow bc, Y \rightarrow Xa\}$, we obtain $S(D) = abcabca$.

For any string, it is uniquely partitioned to $w_1 a_1^+ w_2 a_2^+ \cdots w_k a_k^+ w_{k+1}$ by maximal repetitions, where each a_i is a symbol and w_i is a string containing no repetition. Each a_i^+ is called Type1 metablock, w_i is called Type2 metablock if $|w_i| \geq \log^* n$, and other short w_i is called Type3 metablock, where if $|w_i| = 1$, this is attached to a_{i-1}^+ or a_i^+ , with preference a_{i-1}^+ when both are possible. Thus, any metablock is longer than or equal to two.

Let S be a metablock and D be a current dictionary starting with $D = \emptyset$. We set $ESP(S, D) = (S', D \cup D')$ for $S'(D') = S$ and S' described as follows:

1. When S is Type1 or Type3 of length $k \geq 2$,
 - (a) If k is even, let $S' = t_1 t_2 \cdots t_{k/2}$, and make $t_i \rightarrow S[2i - 1 : 2i] \in D'$.

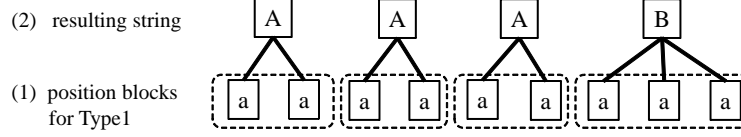


Figure 2: Parsing for Type1 string: Line (1) is an original Type1 string $S = a^9$ with its position blocks. Line (2) is the resulting string AAAB, and the production rules $A \rightarrow aa$ and $B \rightarrow aaa$. Any Type3 string is parsed analogously.

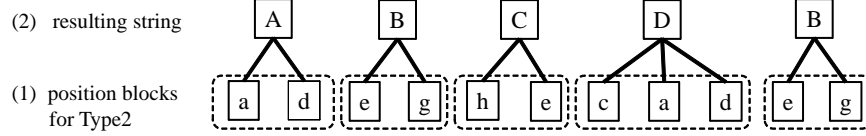


Figure 3: Parsing for Type2 string: Line (1) is an original Type2 string ‘adeghecadeg’ with its position blocks by alphabet reduction where its definition is omitted in this paper. Line (2) is the resulting string ABCDB, and the production rules $A \rightarrow ad$, $B \rightarrow eg$, etc.

- (b) If k is odd, let $S' = t_1 t_2 \cdots t_{(k-3)/2} t$, and make $t_i \rightarrow S[2i-1 : 2i] \in D'$ and $t \rightarrow S[k-2 : k] \in D'$ where t_0 denotes the empty string for $k = 3$.

2. When S is Type2,

- (c) for the partitioned $S = s_1 s_2 \cdots s_k$ ($2 \leq |s_i| \leq 3$) by *alphabet reduction*, let $S' = t_1 t_2 \cdots t_k$, and make $t_i \rightarrow s_i \in D'$.

Cases (a) and (b) denote a typical *left aligned parsing*. For example, in case $S = a^6$, $S' = x^3$ and $x \rightarrow a^2 \in D'$, and in case $S = a^9$, $S' = x^3 y$ and $x \rightarrow a^2, y \rightarrow aaa \in D'$. In Case (c), we omit the description of alphabet reduction [9] because the details are unnecessary in this paper.

Case (b) is illustrated in Fig. 2 for a Type1 string, and the parsing manner in Case (a) is obtained by ignoring the last three symbols in Case (b). Parsing for Type2 is analogous. Case (c) is illustrated in Fig. 3.

Finally, we define ESP for any string $S \in (\Sigma \cup V)^*$ that is partitioned to $S_1 S_2 \cdots S_k$ by k metablocks; $ESP(S, D) = (S', D \cup D') = (S'_1 \cdots S'_k, D \cup D')$, where D' and each S'_i satisfying $S'_i(D') = S_i$ are defined in the above.

Iteration of ESP is defined by $ESP^i(S, D) = ESP^{i-1}(ESP(S, D))$. In particular, $ESP^*(S, D)$ denotes the iterations of ESP until $|S| = 1$. After computing $ESP^*(S, D)$, the final dictionary represents a rooted ordered binary tree deriving S , which is denoted by $ET(S)$.

Lemma 7 (Cormode and Muthukrishnan [9]) The height of $ET(S)$ is $O(\log |S|)$ and $ET(S)$ can be computed in time $O(|S| \log^* |S|)$ time.

Lemma 8 (Cormode and Muthukrishnan [9]) Let $S = s_1 s_2 \cdots s_k$ be the partition of a Type2 metablock S by alphabet reduction. For any $1 \leq j \leq |S|$, the block s_i containing $S[j]$ is determined by at most $S[j - \log^* N - 5 : j + 5]$.

We refer to another characteristic of ESP for pattern embedding problem. Nodes v_1, v_2 in $T = ET(S)$ are *adjacent* in this order if the subtrees on v_1, v_2 are adjacent in this order. A string $p_1 \cdots p_k$ of length k is embedded in T if there exist nodes v_1, \dots, v_k such that $label(v_i) = p_i$ and

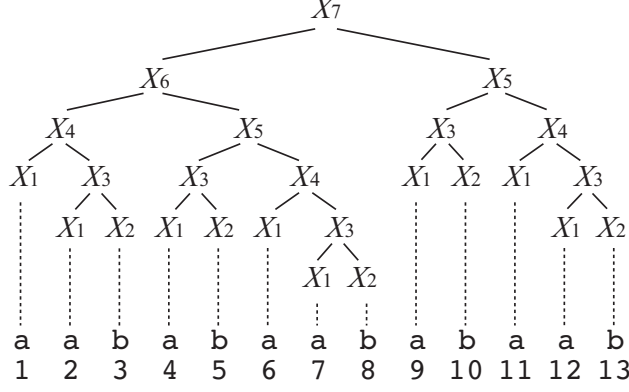


Figure 4: The derivation tree of SLP with $X_1 \rightarrow a$, $X_2 \rightarrow b$, $X_3 \rightarrow X_1X_2$, $X_4 \rightarrow X_1X_3$, $X_5 \rightarrow X_3X_4$, $X_6 \rightarrow X_4X_5$, and $X_7 \rightarrow X_6X_5$, representing string $S = \text{val}(X_7) = \text{aababaababab}$.

any v_i, v_{i+1} are adjacent in this order. If $T[i]$, the i -th leaf of T , is the leftmost leaf of v_1 and $T[j]$ is the rightmost leaf of v_k , we call that $p_1 \cdots p_k$ is embedded as $T[i : j]$.

Definition 9 $Q \in (\Sigma \cup V)^*$ is called an *evidence* of $P \in \Sigma^*$ in S if the following holds: $S[i : j] = P$ iff Q is embedded as $T[i : j]$.

We note that any P has at least one evidence since P itself is an evidence of P .

Lemma 10 (Maruyama et al. [29]) Given $T = ET(S)$, for any $T[i : i + t] = P$, there exists an evidence $Q = q_1 \cdots q_k$ of P with maximal repetitions q_ℓ and $k = O(\log t)$. We can compute the Q in $O(\log t \log |S|)$ time, and we can also check if Q is embedded as $T[j : j + t]$ in $O(\log t \log |S|)$ time for any j .

2.7 Straight Line Programs

A *straight line program* (SLP) [18] is a widely accepted abstract model of outputs of grammar-based compressed methods. An SLP is a sequence of assignments $\{X_i \rightarrow \text{expr}_i\}_{i=1}^n$, where each X_i is a variable and each expr_i is an expression, where $\text{expr}_i = a$ ($a \in \Sigma$), or $\text{expr}_i = X_\ell X_r$ ($\ell, r < i$). Namely, SLPs are admissible grammars in the Chomsky normal form, and hence outputs of admissible grammars can be easily converted to SLPs in linear time (see also Figure 1). Let $\text{val}(X_i)$ represent the string derived from X_i . When it is not confusing, we identify a variable X_i with $\text{val}(X_i)$. Then, $|X_i|$ denotes the length of the string X_i derives. An SLP $\{X_i \rightarrow \text{expr}_i\}_{i=1}^n$ represents the string $S = \text{val}(X_n)$. The *size* of an SLP is the number of assignments in it. The *height* of variable X_i is denoted $\text{height}(X_i)$, and is 1 if $X_i = a$ ($a \in \Sigma$), and $1 + \max\{\text{height}(X_\ell), \text{height}(X_r)\}$ if $X_i = X_\ell X_r$. The height of an SLP $\{X_i \rightarrow \text{expr}_i\}_{i=1}^n$ is defined to be $\text{height}(X_n)$.

Note that $|X_i|$ and $\text{height}(X_i)$ for all variables can be calculated in a total of $O(n)$ time by simple dynamic programming iterations. In the rest of the paper, we will therefore assume that these values will be available.

The following results are known for SLP compressed strings:

Theorem 11 ([25]) *Given two SLPs of total size n that describe strings S and P , respectively, a succinct representation of $\text{Occ}(S, P)$ can be computed in $O(n^3)$ time and $O(n^2)$ space.*

Since we can compute $|S|$ and $|P|$ in $O(n)$ time and a membership query to the succinct representation can be answered in $O(n)$ time [31], the equality checking of whether $S = P$ can be done in a total of $O(n^3)$ time.

Lemma 12 ([24]) *Given an SLP of size n representing string S , an SLP of size $O(n)$ which represents an arbitrary substring $S[i : j]$ can be constructed in $O(n)$ time.*

3 Algorithms for Restructuring Compressed Texts

In this section we present our polynomial-time algorithms that converts an input compressed representation to another compressed representation. In the sequel, n and m will denote the sizes of the input and output compressed representations, respectively.

3.1 Conversions from Run Length Encoding

For conversions from Run Length Encodings, we obtain the results below.

Theorem 13 (Run Length Encoding to Re-pair) *Given an RL factorization of size n that represents string S , the grammar of size m produced by applying Re-pair algorithm to S can be computed in $O(nm \log m)$ time.*

Proof. We consider a simple simulation of the Re-pair algorithm that works on the RL factorization of the string S . We shall assume that the Re-pair algorithm replaces non-overlapping bigrams with a new variable in a left-first manner. Let $Y_i \rightarrow Y_\ell Y_r$ denote the i -th rule produced by the Re-pair algorithm running on S . Let $S_1 = S$, and for $i \geq 1$ let S_{i+1} denote the string obtained by replacing frequent bigrams by Y_1, Y_2, \dots , and Y_i . Note that the bigram $Y_\ell Y_r$ will not occur in S_{i+1} . Consider the RL factorization of S_i , and let w_i denote the string obtained by concatenating the RL factors of S_i consisting of characters in $\Sigma \cup \{Y_j\}_{j=1}^{i-1}$.

We find the most frequent bigram $Y_\ell Y_r$ in w_i , and then replace non-overlapping occurrence of $Y_\ell Y_r$ in w_i with a new variable Y_i on the left priority basis, and then compute w_{i+1} .

Let $a, b, c \in \Sigma$ with $a \neq b$ and $a \neq c$, and let $ba^p c$ be a substring of the original string S , where $p \geq 1$. Consider any occurrence of $ba^p c$ that begins at position v in S , namely, let $S[v : v+p+1] = ba^p c$. There are two cases to consider: (1) the range $[v : v+p+1]$ is fully contained within a variable Y_k in w_i ; (2) the range $[v : v+p+1]$ is contained in a substring of w_i of form $(Y_s)^e (Y_{k(1)})^q Y_{k(2)} \cdots Y_{k(l)} (Y_r)^t$ with $k(1) > k(2) > \cdots > k(l)$, where $\text{val}(Y_{k(1)})^q \cdot \text{val}(Y_{k(2)}) \cdots \text{val}(Y_{k(l)}) = a^{p'}$ for some $p' \leq p$, ba^x is a suffix of $\text{val}(Y_s)$, $a^y c$ is a prefix of $\text{val}(Y_r)$, and $x + p' + y = p$.

Let $Y_\ell Y_r$ be the most frequent bigram in w_i . It is possible to replace non-overlapping occurrences of $Y_\ell Y_r$ in $O(n)$ time, as follows: We can see that $\text{val}(Y_\ell)\text{val}(Y_r)$ occurs either (A) in a sequence $f_j f_{j+1} \cdots f_{j+d-1}$ of $d \geq 2$ consecutive factors in w_1 or (B) entirely within a single factor f_j of w_1 . This is because, if $\text{val}(Y_\ell)\text{val}(Y_r)$ contains at least two distinct characters $a \neq b$, then it occurs in a sequence of d factors, and if $\text{val}(Y_\ell)\text{val}(Y_r) = a^z$, then it is fully contained in a factor. Consider case (A): Let $\text{val}(Y_\ell)[1] = c$. Since the number of factors of form $f = c^p$ does not exceed n , the number of occurrences of the bigram of case (A) is $O(n)$. Now consider case (B): According to the observation (2) above, any bigram $Y_\ell Y_r$ with $\text{val}(Y_\ell)\text{val}(Y_r) = a^z$ and $\ell \neq r$ occurs at most once in each substring of w_i that corresponds to a factor f_j . Hence the number of occurrences of such a bigram in w_i is at most n . If $\ell = r$, then the bigram $Y_\ell Y_\ell$ can occur $q - 1$ times at each factor. We then replace $(Y_\ell)^q$ with $(Y_i)^{q/2}$ if q is even, and with $(Y_i)^{(q-1)/2} Y_\ell$ otherwise, in $O(1)$ time.

Since each w_i consists of characters in $\Sigma \cup \{Y_j\}_{j=1}^{i-1}$, the number of all bigrams in w_i is $O(m^2 + m|\Sigma|) = O(m^2)$. We find the most frequent bigram in $O(\log m)$ time using a heap, and the total

time complexity for converting the RL factorization to the grammar corresponding to Re-pair is $O(nm \log m)$. ■

Theorem 14 (Run Length Encoding to LZ77/LZ88) *Given an RL factorization of size n that represents string S , the LZ factorization of S can be computed in $O(nm + n \log n)$ time, where m is the size of LZ factorization.*

Proof. Let $a_1^{p_1}, \dots, a_n^{p_n}$ be the RL factorization of a text S . Assume that we have already computed the first $i - 1$ LZ77 factors, f_1, \dots, f_{i-1} , of S . Let the pair of integers (u, q) satisfy $q + p_1 + \dots + p_{u-1} = |f_1 \dots f_{i-1}|$, where $1 \leq u \leq n$ and $1 \leq q \leq p_u$. For a new factor f_i , compute the lengths l_j of the longest common prefix of $a_{u+1}^{p_{u+1}} \dots a_n^{p_n}$ and each suffix $a_j^{p_j} \dots a_n^{p_n}$ ($1 \leq j \leq u$) of the RLE, where each RL factor a^p is regarded as single symbol. The length of the i -th LZ77 factor is then: $\max_j \{p_{u+1} + \dots + p_{u+l_j} + P + Q\}$, where $P = 0$ if $a_u \neq a_j$ and $P = \min\{p_u - q, p_{j-1}\}$ otherwise, and $Q = 0$ if $a_{u+l_j+1} \neq a_{j+l_j}$ and $Q = \min\{p_{u+l_j+1}, p_{j+l_j}\}$ otherwise. The process is then repeated to obtain f_{i+1} from the pair of integers $(u + l_j + 1, p_{u+l_j+1} - Q)$. A naïve algorithm for obtaining each l_j costs $O(n)$ time, and therefore results in an $O(n^2m)$ time algorithm to check each of the $O(n)$ suffixes to construct the $O(m)$ factors. If we construct a suffix and lcp array on the RLE string beforehand, l_j can be computed in $O(1)$ time, since it amounts to a range minimum query on the lcp array. Note that the sum $p_{u+1} + \dots + p_{u+l_j}$ can also be obtained in constant time with $O(n)$ preprocessing, by constructing an array $sum[i] = p_1 + \dots + p_i$ and computing $sum[u + l_j] - sum[u]$. Therefore, conversion can be done in $O(nm)$ time provided that the suffix array and lcp arrays are constructed. The construction of the arrays require $O(n \log n)$ time, to sort and number each of character of the alphabet $a_i^{p_i}$. ■

LZ78 factorization can be achieved by a simple modification. ■

Theorem 15 (Run Length Encoding to Bisection) *Given an RL factorization of size n that represents string S , the grammar of size m produced by applying Bisection algorithm to S can be computed in $O(m^2 + (m + n) \log n)$ time.*

Proof. Consider the following top-down algorithm which closely follows the description of Bisection in Section 2.5.2. Assume we want to construct the children Y_ℓ, Y_r of variable Y_s representing $S[i : j]$, to produce the grammar rule $Y_s \rightarrow Y_\ell Y_r$. Note that an arbitrary substring of $S[i : j]$ which is contained in the RLE $a_{k-1}^{p_{k-1}} \dots a_{k+l}^{p_{k+l}}$ can be represented as a 4-tuple (x, k, l, y) , where $i = p_1 + \dots + p_{k-1} - x + 1$, $j = p_1 + \dots + p_{k+l-1} + y - 1$, $0 \leq x < p_{k-1}$, and $0 \leq y < p_{k+l}$. Let k represent the largest integer where $2^k < j - i + 1$. For the substring $S[i : j]$ under consideration, the 4-tuple for substrings $S[i : i + 2^k - 1]$ and $S[i + 2^k : j]$ can be obtained in $O(\log n)$ time. Note that equality checks between substrings represented as 4-tuples can be conducted in $O(1)$ time with $O(n \log n)$ preprocessing, using range minimum queries on the lcp arrays, similar to the technique used in the conversion to LZ encodings. Equality checks are conducted against the $O(m)$ variables that will be contained in the output. If there exist variables which derive the same string, the existing variables are used in place of Y_ℓ and/or Y_r , and Y_ℓ and/or Y_r will not be contained in the output. Since equality checks are conducted only for the children of variables which are contained in the output, they are conducted only $O(m)$ times. Therefore, conversion can be done in $O(n \log n + m(m + \log n)) = O(m^2 + (m + n) \log n)$ time. ■

3.2 Conversions from arbitrary SLP

Theorem 16 (SLP to Run Length Encoding) *Given an SLP of size n that represents string S , the RL factorization of S can be computed in $O(n + m)$ time and $O(n)$ space, where m is the size of the RL factorization.*

Proof. For each variable X_i , we first compute the maximal length of the run of identical characters which is a prefix (resp. suffix) of X_i , denoted by $plen(X_i)$ (resp. $slen(X_i)$). This can be computed in $O(n)$ time by a simple dynamic programming: for $X_i \rightarrow X_\ell X_r$, we have $plen(X_i) = plen(X_\ell)$ if $plen(X_\ell) < |X_\ell|$ or $X_\ell[|X_\ell|] \neq X_r[1]$, and $plen(X_i) = plen(X_\ell) + plen(X_r)$ otherwise. $slen(X_i)$ can be computed likewise.

Next, for each variable $X_i \rightarrow X_\ell X_r$, let $Llink(X_i)$ denote the variable $X_{i'} \rightarrow X_{\ell'} X_{r'}$ such that $X_{i'}$ is the shallowest descendant of X_i lying on the left most path of the derivation tree of X_i , satisfying $slen(X_{i'}) \leq |X_{r'}|$. $Llink(X_i)$ can also be computed for all X_i in $O(n)$ time, by a simple dynamic programming. $Rlink(X_i)$ can be defined and computed likewise.

The conversion algorithm is then a top down post-order traversal on the derivation tree of SLP but with jumps using $Llink$ and $Rlink$. For the root X_n , we output (1) $X_n[1]^{plen(X_n)}$, (2) the RLE of X_n except for the first and last RL factors of X_n , and (3) $X_n[|X_n|]^{slen(X_n)}$. (2) can be computed recursively as follows: at each variable $X_i \rightarrow X_\ell X_r$, we output (2.1) the RLE of $Llink(X_i)$ except for the first and last RL factors of $Llink(X_i)$, (2.2) either $X_\ell[|X_\ell|]^{slen(X_\ell)} X_r[1]^{plen(X_r)}$ if $X_\ell[|X_\ell|] \neq X_r[1]$, or $X_r[1]^{slen(X_\ell) + plen(X_r)}$ if $X_\ell[|X_\ell|] = X_r[1]$, and (2.3) the RLE of $Rlink(X_i)$ except for the first and last RL factors of $Rlink(X_i)$. The theorem follows since the output of each RL factor is done in constant time. \blacksquare

Theorem 17 (SLP to LZ77) *Given an SLP of size n that represents string S , the LZ77 factorization of size m can be computed in $O(mn^3 \log N)$ time.*

Proof. Assume we have already computed f_1, \dots, f_{i-1} of S from a given SLP of size n . Firstly we consider the LZ77 factorization without self-references. For a new factor f_i , do a binary search on the length of the factor: create a new SLP of that length, and conduct pattern matching on the input SLP. If a match exists in the range that corresponds to the previous factors f_1, \dots, f_{i-1} , i.e., in the prefix $S[1 : \sum_{j=1}^{i-1} |f_j|]$ of S , then the length of f_i can be longer, and if not, it must be shorter. Using Theorem 11 and Lemma 12 the LZ77 factorization of size m can thus be computed in $O(mn^3 \log N)$ time. To compute the LZ77 factorization with self-references, we search for the longest match that begins at a position from 1 to $\sum_{j=1}^{i-1} |f_j|$ in S . The time complexity is the same as above. \blacksquare

Theorem 18 (SLP to LZ78) *Given an SLP of size n that represents string S , the LZ78 factorization of size m can be computed in $O(n^3 m^2)$ time.*

Proof. Our algorithm for converting an SLP to LZ78 follows a similar idea: When computing a new factor f_i , we construct a new SLP of $f_k f_{k+1}[1]$ for each $1 \leq k < i$, and run the pattern matching algorithm on the input SLP. The longest match in the suffix $S[\sum_{j=1}^{i-1} |f_j| + 1 : |S|]$ provides the new factor f_i . By Theorem 11 and Lemma 12, pattern matching tasks for computing each factor f_i takes $O(n^3 m)$ time, and therefore the total time complexity is $O(n^3 m^2)$. \blacksquare

3.3 SLP to Bisection

Theorem 19 *Given an SLP of size n that represents string S , the grammar of size m produced by applying Bisection algorithm to S can be computed in $O(n^3m^2)$ time.*

Proof. Given an arbitrary SLP of size n representing S , consider the following top-down algorithm which closely follows the description of Bisection in Section 2.5.2. Assume we want to construct the children Y_ℓ, Y_r of variable Y_s representing $S[i : j]$, to produce the grammar rule $Y_s \rightarrow Y_\ell Y_r$. Let k represent the largest integer where $2^k < j - i + 1$. By using Lemma 12, SLPs Y_ℓ representing $S[i : i + 2^k - 1]$ and Y_r representing $S[i + 2^k : j]$, can be constructed in $O(n)$ time. For these SLPs, equality checks are conducted against all $O(m)$ variables corresponding to variables that will be contained in the output produced so far. If there exist variables which derive the same string, the existing variables are used in place of Y_ℓ and/or Y_r , and Y_ℓ and/or Y_r will not be contained in the output. From Theorem 11, the equality checks for Y_ℓ and Y_r can be conducted in a total of $O(n^3m)$ time. Since equality checks are conducted only for the children of variables which are contained in the output, the total time is $O(n^3m^2)$. \blacksquare

3.4 Conversions to and from ESP

Given a representation of SLP G for a string S , we design algorithms to compute LZ77 and LZ78 factorizations for S without explicit decompression of G in $O((n + m) \log^d N)$ time. Here n/m is the size of input/output grammar size, $N = |S|$, and d is a constant. Our method is based on the transformation of any SLP to its canonical form by way of an equivalent ESP.

Lemma 20 *Given a dictionary D from $ESP^*(S, D)$ for some $S \in \Sigma^*$, and the set V of variables in D , we can compute an SLP with the dictionary D' and the set V' of variables which satisfies the following conditions: (1) $|D'| \leq 2|D|$ and (2) for any $X_i, X_j \in V'$, $val(X_i) \leq_{lex} val(X_j)$ iff $i \leq j$, where \leq_{lex} denotes the lexical order over Σ . The computation time is $O(n \log n \log^3 N)$ for $|V| = n$ and $|S| = N$.*

Proof. Consider $T_X = ET(val(X))$ and $T_Y = ET(val(Y))$ for any $X, Y \in V$. Let $t = \lfloor |val(Y)|/2 \rfloor$. By Lemma 10, we can compute an evidence Q of the pattern $T_Y[1 : t]$ in $O(\log^2 t) = O(\log^2 N)$ time. We can also check if Q is embedded as $T_X[1 : t]$ in $O(\log^2 N)$ time. By this binary search, we can find the length of longest common prefix of $val(X)$ and $val(Y)$ in $O(\log^2 N)$ time. Thus, we can sort all variables in V in $O(n \log n \log^3 N)$ time. After sorting all variables in V , we rename any variable according to its rank. If there is a variable X with $X \rightarrow X_i X_j X_k$, we divide it to $X \rightarrow Y X_k$ and $Y \rightarrow X_i X_j$ by an intermediate variable Y and we can determine the rank of such new variables in additional $O(n \log n)$ time. \blacksquare

Dictionaries D_1, D_2 of two admissible grammars are called *consistent* if $X \rightarrow \alpha, Y \rightarrow \alpha \in D_1 \cup D_2$ implies $X = Y$, and consistent dictionaries D_1, \dots, D_k are similarly defined.

For $\alpha \in (\Sigma \cup V)^*$, $\alpha = q_1 \cdots q_k$ is called a *run-length representation* of α if each q_i is a maximal repetition of $p_i \in \Sigma \cup V$. For example, the run-length representation of *abbaaaca* is $q_1 q_2 q_3 q_4 q_5 = ab^2 a^3 ca^2$. The number k of $\alpha = q_1 \cdots q_k$ is called the *change* of α .

Let $S = \alpha\beta\gamma$ and $S' = \alpha'\beta'\gamma'$ satisfying $ESP(S, D) = (S', D \cup D')$ with $\alpha'(D') = \alpha$, $\beta'(D') = \beta$, and $\gamma'(D') = \gamma$. Then we call such $S = \alpha\beta\gamma$ a *stable decomposition* of S . An expression $ESP(\alpha[\beta]\gamma, D) = (\alpha'[\beta']\gamma', D \cup D')$ denotes an ESP to replace the $\alpha/\beta/\gamma$ to the $\alpha'/\beta'/\gamma'$, respectively. For a string α , $\overline{\alpha}$ and $\underline{\alpha}$ denote a prefix of α and a suffix of α , respectively.

Lemma 21 *Let $ESP(\alpha[\beta]\gamma, D) = (\alpha'[\beta']\gamma', D \cup D')$ for a stable decomposition $S = \alpha\beta\gamma$. There exist substrings $\underline{\alpha}$, $\underline{\alpha\beta}$, $\overline{\beta\gamma}$, $\overline{\gamma}$, each of whose change is at most $\log^*|S| + 5$ such that*

$$\begin{aligned} ESP([\alpha]\overline{\beta\gamma}, D) &= ([\alpha']y_1, D \cup D_1), \\ ESP(\underline{\alpha}[\beta]\overline{\gamma}, D) &= (x_2[\beta']y_2, D \cup D_2), \\ ESP(\underline{\alpha\beta}[\gamma], D) &= (x_3[\gamma'], D \cup D_3), \text{ and} \\ D' &= D_1 \cup D_2 \cup D_3. \end{aligned}$$

Proof. Since $S = \alpha\beta\gamma$ is a stable decomposition of an ESP for S , the translated string α' and the dictionary D_1 for $D_1(\alpha') = \alpha$ are determined by only α and a prefix $\overline{\beta\gamma}$. In case p^+ is the maximal prefix of $\beta\gamma$, we can set $\overline{\beta\gamma} = p^+$. Otherwise, by Lemma 8, we can set $\overline{\beta\gamma}$ to be a prefix of length at most $\log^*|S| + 5$. For β, γ , we can set $\underline{\alpha\gamma}, \underline{\alpha\beta}$ with the bounded change, respectively. The above ESP defines $\alpha'(D_1) = \alpha$, $\beta'(D_2) = \beta$, and $\gamma'(D_3) = \gamma$. By renaming all variables in the dictionaries, there is a consistent $D' = D_1 \cup D_2 \cup D_3$ satisfying $\alpha'(D')\beta'(D')\gamma'(D') = \alpha\beta\gamma$. ■

Lemma 22 *Let D be a dictionary of an SLP encoding a string $S \in \Sigma^*$. A dictionary D' of an ESP equivalent to D is computable in $O(n \log^2 N + m)$ time, where $n = |D|$, $m = |D'|$, and $N = |S|$.*

Proof. We assume that $ESP^*(val(X_\ell), D)$ ($\ell \leq i, j$) is already computed and let D' be the current dictionary consistent with all $val(X_\ell)$. For $X_k \rightarrow X_i X_j$ ($k > i, j$), we estimate the time to update D' . Let $val(X_i) = \alpha$ and $val(X_j) = \gamma$.

For the initial strings α, γ , we can obtain $\underline{\alpha}$ of length $\log^*N + 6$ and $\overline{\gamma}$ of length 6 in $O(\log N \log^*N)$ time. By the result of $ESP(\underline{\alpha}\overline{\gamma}, D')$, we determine the position block β which $\alpha[|\alpha|]$ and $\gamma[1]$ belong to. Then we can find a stable decomposition $S = \alpha\beta\gamma$ for the obtained β and reformed α and γ , where α (and γ) is represented by a path from the root to a leaf in the derivation tree of D_x (and D_y). They are called a current tail and head, respectively. Note that we can avoid decoding α and γ for the parsing in Lemma 21. To simulate this, we use only the compressed representations D_x, D_y , the current tail/head, and β . Using the run-length representation, the change of β is bounded by $O(\log^*N)$ as follows.

By Lemma 21, when $ESP(\alpha[\beta]\gamma, D) = (\alpha'[\beta']\gamma', D \cup D')$ is computed by $ESP([\alpha]\overline{\beta\gamma}, D) = ([\alpha']y_1, D \cup D_1)$, $ESP(\underline{\alpha}[\beta]\overline{\gamma}, D) = (x_2[\beta']y_2, D \cup D_2)$, and $ESP(\underline{\alpha\beta}[\gamma], D) = (x_3[\gamma'], D \cup D_3)$, the resulting string β' is treated as the next β , and the current tail and head are replaced by $\alpha'[[\alpha']]$ and $\gamma'[1]$ which represent the next α and γ .

Let us consider the case $ESP([\alpha]\overline{\beta\gamma}, D) = ([\alpha']y_1, D \cup D_1)$. If $\overline{\beta\gamma}$ contains a maximal repetition of p as $\alpha[|\alpha| - N_1, |\alpha| \cdot \overline{\beta\gamma}[1, N_2] = p^+$, the next tail is the parent of $\alpha[|\alpha| - N_1 - 1]$, which is determined in $O(\log^2 N)$ time since any repetition is replaced by the left aligned parsing and $N_1 + N_2 = O(N)$. Otherwise, by Lemma 8, we can determine the next tail by tracing a suffix of α of length at most $\log^*N + 5$ in $O(\log N \log^*N)$ time.

Thus, $ESP(\alpha[\beta]\gamma, D) = (\alpha'[\beta']\gamma', D \cup D')$ is simulated in $O(\log^2 N + m_k)$ for $X_k \rightarrow X_i X_j$, where m_k is the number of new variables produced in this ESP. Therefore we conclude that the final dictionary D' equivalent to D is obtained in $O((\log^2 N + m_1) + \dots + (\log^2 N + m_n)) = O(n \log^2 N + m)$. ■

Theorem 23 (SLP to Canonical SLP) *Given an SLP D of size n for string S of length N , we can construct another SLP D' of size m in $O(n \log^2 N + m \log m \log^3 N)$ such that D' is a final dictionary of $ESP^*(S, D')$ equivalent to D and all variables in D' are sorted by the lexical order of their encoded strings.*

Theorem 24 (Canonical SLP to LZ77) Given a canonical SLP D of size n for string S of length N , we can compute LZ77 factorization f_1, \dots, f_m of S in $O(m \log^2 n \log^3 N + n \log^2 n)$ time.

Proof. Using the technique in Lemma 20, we can sort all variables Z associated with $Z \rightarrow XY \in D$ by the following two keys: the first key is the lexical order of $\text{val}(X)^R$ and the second is the lexical order of $\text{val}(Y)$, where S^R denotes the reverse string of $S \in \Sigma^*$. Then Z is mapped to a point (i, j, pos) on a 3-dimensional space such that i is an index of first key on X -axis, j is an index of second key on Y -axis, and pos is an index of leftmost occurrence of $\text{val}(Z)$ on Z -axis. A data structure supporting range query for the point set is constructed in $O(n \log^2 n)$ time/space and achieving $O(\log^2 n)$ query time (See [26]). Using this, we can compute $f_{\ell+1}$ from f_1, \dots, f_ℓ and the remaining suffix S' such that $S = f_1 \cdots f_\ell \cdot S'$ as follows.

By Lemma 10, an evidence $Q = q_1 \cdots q_k$ of $S'[1 : j]$ satisfying $k = O(\log j)$ is found in $O(\log^2 j)$ time. Let q_i be a symbol. Then we guess the division $\alpha = q_1 \cdots q_i$ and $\beta = q_{i+1} \cdots q_k$ to find the range of X in which α is embedded as its suffix, the range of Y in which β is embedded as its prefix, and the range of Z whose leftmost position pos satisfies $\text{pos} + j \leq |f_1 \cdots f_\ell|$. This query time is $O(\log^2 n \log^2 N)$. Let $q_i = p_i^j$ for a symbol p_i . Any maximal repetition is replaced by the left aligned parsing, and a resulting new repetition is recursively replaced by the same manner. Thus, an embedding of $q_1 \cdots q_k$ to $Z \rightarrow XY$ dividing $q_i = \alpha\beta$ such that $q_1 \cdots q_{i-1}\alpha$ is embedded to X as suffix and $\beta q_{i+1} \cdots q_k$ is embedded to Y as prefix is possible in $O(\log j) = O(\log N)$ divisions for q_i . In this case, the query time is $O(\log^2 n \log^3 N)$. Therefore, the total time to compute the required LZ77 factorization is bounded by $O(m \log^2 n \log^3 N + n \log^2 n)$. ■

Theorem 25 (Canonical SLP to LZ78) Given a canonical SLP D of size n for string S of length N , we can compute LZ78 factorization f_1, \dots, f_m of S in $O(m \log^3 N + n)$ time.

Proof. Assume that the first ℓ factors f_1, \dots, f_ℓ are obtained. By Lemma 10, we can find an evidence Q_i of f_i ($1 \leq i \leq \ell$), and all evidences Q_i ($1 \leq i \leq \ell$) are represented by a trie. Let S' be the remaining suffix of S . For each j , we can compute an evidence of $S'[1 : j]$ in $O(\log^2 j) = O(\log^2 N)$ time. Thus, we can find the greatest j satisfying $f_i = S'[1 : j]$ for some $1 \leq i \leq \ell$ in $O(\log^3 N)$ time using binary search. Therefore, the total time to compute the required LZ78 factorization is bounded by $O(m \log^3 N + n)$. ■

Theorem 26 (Run Length Encoding to ESP) Given a text S represented as a RL encoding $S = f_1 \cdots f_n$ of length n , we can compute an ESP D representing S in $O(n \log^* N)$ time.

Proof. We make a little change for replacing maximal repetition. Consider maximal repetition $\alpha = a^k$ in S is appeared. If k is even, then we replace α to $A^{k/2}$, otherwise we replace to $A^{\lfloor k/2 \rfloor - 1} B$ where $A \rightarrow aa$ and $B \rightarrow aaa$. In exceptional case that the prefix and/or suffix of α is replaced with the left/right symbol adjacent to α , we must consider for α' removed such prefix/suffix from α . The computation time to replace such repetition is $O(1)$ since the number of repetitive symbols is represented as a integer. Therefore, the time to convert is bounded by $O(n \log^* N)$. ■

Theorem 27 (ESP to Bisection) Given an ESP D of size n representing S of length N , we can compute an SLP of size m generated by Bisection in $O(m \log^2 N)$ time.

Proof. For each corresponding substring $S[i : j]$ under consideration. We can obtain an evidence Q corresponding to $S[i : i + 2^k - 1]$ in $O(\log^2 N)$ time. By the Lemma 10, we can check if Q is embedded as $T[i + 2^k : j]$ in $O(\log^2 N)$ time. If Q can be embedded, we can allocate same variable for $S[i : i + 2^k - 1]$ and $S[i + 2^k : j]$ since both substrings are equal, otherwise different variables are allocated. Therefore, conversion can be done in $O(m \log^2 N)$ time. ■

4 Conclusions and Future Work

In this paper we presented new efficient algorithms which, without explicit decompression, convert to/from compressed strings represented in terms of run length encoding (RLE), LZ77 and LZ78 encodings, grammar based compressor RE-PAIR and BISECTION, edit sensitive parsing (ESP), straight line programs (SLPs), and admissible grammars. All the proposed algorithms run in polynomial time in the input and output sizes, while algorithms that first decompress the input compressed string can take exponential time. Examples of applications of our result are dynamic compressed strings allowing for edit operations, and post-selection of specific compression formats.

Future work is to extend our results to other text compression schemes, such as Sequitur [34], Longest-First Substitution [32], and Greedy [1].

References

- [1] APOSTOLICO, A., AND LONARDI, S. Off-line compression by greedy textual substitution. *Proc. IEEE* 88 (2000), 1733–1744.
- [2] BILLE, P., LANDAU, G. M., RAMAN, R., SADAKANE, K., SATTI, S. R., AND WEIMANN, O. Random access to grammar-compressed strings. In *Proc. SODA '11* (2011), pp. 373–389.
- [3] CHAN, H.-L., HON, W.-K., LAM, T.-W., AND SADAKANE, K. Dynamic dictionary matching and compressed suffix trees. In *Proc. SODA '05* (2005), pp. 13–22.
- [4] CHAN, H.-L., HON, W.-K., LAM, T.-W., AND SADAKANE, K. Compressed indexes for dynamic text collections. *ACM Trans. Algorithms* 3, 2 (2007), 21:1–21:29.
- [5] CHARIKAR, M., LEHMAN, E., LIU, D., PANIGRAHY, R., PRABHAKARAN, M., SAHAI, A., AND SHELAT, A. The smallest grammar problem. *IEEE Transactions on Information Theory* 51, 7 (2005), 2554–2576.
- [6] CILIBRASI, R., AND VITÁNYI, P. M. B. Clustering by compression. *IEEE Transactions on Information Theory* 51 (2005), 1523–1545.
- [7] CLAUDE, F., AND NAVARRO, G. Self-indexed grammar-based compression. *Fundamenta Informaticae* (to appear). Preliminary version: Proc. MFCS 2009 pp. 235–246.
- [8] CORMODE, G., AND MUTHUKRISHNAN, S. Substring compression problems. In *Proc. SODA '05* (2005), pp. 321–330.
- [9] CORMODE, G., AND MUTHUKRISHNAN, S. The string edit distance matching problem with moves. *ACM Trans. Algor.* 3, 1 (2007), Article 2.
- [10] GAWRYCHOWSKI, P. Optimal pattern matching in LZW compressed strings. In *Proc. SODA '11* (2011), pp. 362–372.
- [11] GAWRYCHOWSKI, P. Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. In *Proc. ESA2011* (2011). accepted (available as arXiv:1104.4203v1).
- [12] GĄSIENIEC, L., KARPINSKI, M., PLANDOWSKI, W., AND RYTTER, W. Efficient algorithms for Lempel-Ziv encoding. In *Proc. SWAT 1996* (1996), vol. 1097 of *LNCS*, pp. 392–403.

- [13] GONZÁLEZ, R., AND NAVARRO, G. Rank/select on dynamic compressed sequences and applications. *Theoretical Computer Science* 410 (2009), 4414–4422.
- [14] GOTO, K., BANNAI, H., INENAGA, S., AND TAKEDA, M. Towards efficient mining and classification on compressed strings. In *Accepted for SPIRE'11* (2011). Preprint available at arXiv:1103.3114v1.
- [15] HERMELIN, D., LANDAU, G. M., LANDAU, S., AND WEIMANN, O. A unified algorithm for accelerating edit-distance computation via text-compression. In *Proc. STACS 2009* (2009), pp. 529–540.
- [16] INENAGA, S., AND BANNAI, H. Finding characteristic substring from compressed texts. In *Proc. The Prague Stringology Conference 2009* (2009), pp. 40–54.
- [17] KÄRKKÄINEN, J., AND SANDERS, P. Simple linear work suffix array construction. In *Proc. ICALP 2003* (2003), vol. 2719 of *LNCS*, pp. 943–955.
- [18] KARPINSKI, M., RYTTER, W., AND SHINOHARA, A. An efficient pattern-matching algorithm for strings with short descriptions. *Nordic Journal of Computing* 4 (1997), 172–186.
- [19] KASAI, T., LEE, G., ARIMURA, H., ARIKAWA, S., AND PARK, K. Linear-time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proc. CPM 2001* (2001), vol. 2089 of *LNCS*, pp. 181–192.
- [20] KIEFFER, J., AND YANG, E. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory* 46, 3 (2000), 737–754.
- [21] KIEFFER, J., YANG, E., NELSON, G., AND COSMAN, P. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory* 46, 4 (2000), 1227–1245.
- [22] KREFT, S., AND NAVARRO, G. Self-indexing based on LZ77. In *Proc. CPM'11* (2011), vol. 6661 of *LNCS*, pp. 41–54.
- [23] LEE, S., AND PARK, K. Dynamic rank/select structures with applications to run-length encoded texts. *Theoretical Computer Science* 410, 43 (2009), 4402–4413.
- [24] LIFSHITS, Y. Solving classical string problems on compressed texts. In *Combinatorial and Algorithmic Foundations of Pattern and Association Discovery* (2006), no. 06201 in Dagstuhl Seminar Proceedings.
- [25] LIFSHITS, Y. Processing compressed texts: A tractability border. In *Proc. CPM 2007* (2007), vol. 4580 of *LNCS*, pp. 228–240.
- [26] LUEKER, G. A data structures for orthogonal range queries. In *Proc. FOCS'78* (1978), pp. 28–34.
- [27] MÄKINEN, V., AND NAVARRO, G. Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algorithms* 4, 3 (2008), 32:1–32:38.
- [28] MANBER, U., AND MYERS, G. Suffix arrays: A new method for on-line string searches. *SIAM J. Computing* 22, 5 (1993), 935–948.

- [29] MARUYAMA, S., NAKAHARA, M., KISHIUE, N., AND SAKAMOTO, H. ESP-Index: A compressed index based on edit-sensitive parsing. In *SPIRE'11* (2011). accepted (available from <http://hdl.handle.net/2324/19843>).
- [30] MATSUBARA, W., INENAGA, S., ISHINO, A., SHINOHARA, A., NAKAMURA, T., AND HASHIMOTO, K. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theoret. Comput. Sci.* 410, 8–10 (2009), 900–913.
- [31] MIYAZAKI, M., SHINOHARA, A., AND TAKEDA, M. An improved pattern matching algorithm for strings in terms of straight-line programs. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM '97)* (1997), vol. 1264 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–11.
- [32] NAKAMURA, R., INENAGA, S., BANNAI, H., FUNAMOTO, T., TAKEDA, M., AND SHINOHARA, A. Linear-time off-line text compression by longest-first substitution. *Algorithms* 2, 4 (2009), 1429–1448.
- [33] NAVARRO, G., AND MÄKINEN, V. Compressed full-text indexes. *ACM Computing Surveys* 39, 1 (2007), 2.
- [34] NEVILL-MANNING, C. G., WITTEN, I. H., AND MAULSBY, D. L. Compression by induction of hierarchical grammars. In *Proc. Data Compression Conference 1994 (DCC '94)* (1994), pp. 244–253.
- [35] RUSSO, L. M. S., NAVARRO, G., AND OLIVEIRA, A. L. Dynamic fully-compressed suffix trees. In *Proc. CPM'08* (2008), vol. 5029 of *LNCS*, pp. 191–203.
- [36] RYTTER, W. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoret. Comput. Sci.* 302, 1–3 (2003), 211–222.
- [37] STORER, J., AND SZYMANSKI, T. Data compression via textual substitution. *Journal of the ACM* 29, 4 (1982), 928–951.
- [38] TISKIN, A. Towards approximate matching in compressed strings: Local subsequence recognition. In *Proc. CSR'11* (2011), vol. 6651 of *LNCS*, pp. 410–414.
- [39] WELCH, T. A. A technique for high performance data compression. *IEEE Computer* 17 (1984), 8–19.
- [40] YAMAMOTO, T., BANNAI, H., INENAGA, S., AND TAKEDA, M. Faster subsequence and don't-care pattern matching on compressed texts. In *Proc. CPM'11* (2011), vol. 6661 of *LNCS*, pp. 309–322.
- [41] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* IT-23, 3 (1977), 337–343.
- [42] ZIV, J., AND LEMPEL, A. Compression of individual sequences via variable-length coding. *IEEE Transactions on Information Theory* 24, 5 (1978), 530–536.